Lecture 2: Python Statements

IPEC Winter School 2015
B-IT

Dr. Tiansi Dong & Dr. Joachim Köhler
Structure of Python Programs

- A Python **program** consists of several **modules**
- A Python **module** consists of **statements**
- A Python **statement** consists of **expressions**
- A Python **expression** creates or processes Python **objects**
Assignment statement 1

- Assignment creates an object reference
  
  ```
  >>>var = 123
  >>>var0, var1 = 1234, 'abcd'
  >>>a, b, c, d = '1234'
  >>>x, y, z = 123
  ```

- Sequence assignments, when the left and the right sides of '==' have the same number of elements
  
  ```
  >>>a, b, c = list(var1[:2]) + [var1[2:]]
  >>>a, b, c = list(var1[:2]), [var1[2:]]
  >>>((a, b), c) = ('12', '34')
  ```
Extended sequence assignment in Python 3.0: On the left side of '="', a starred name can match more than one elements

```python
>>=[a,*b,c] = '1234'
>>>b
['2', '3']
>>=[a,*b,*c, d] = '1234'
>>=[a,b,c,d,*e] = '1234'
>>>e
[]
>>=[a,b,*c,d,e] = '1234'
>>=a,b,c,d,e
('1', '2', [], '3', '4')
>>>*a = 'asdf'
>>> [*a] = 'adsf'
>>> a
['a', 'd', 's', 'f']
>>> *a, = 'adsf'
>>> a
['a', 'd', 's', 'f']
```
Assignment statement 3

- Multiple-target assignments, all the targets point to the same object
  
  >>> a = b = c = []
  >>> a, b, c
  ([], [], [])
  >>> b.append(12)
  ([12], [12], [12])
  >>> c = [1, 2, 3]
  >>> a, b, c
  ([12], [12], [1, 2, 3])

- Augmented assignments
  
  >>> X = 1
  >>> X += 1
  >>> X
  2
Assignment statement 4

• Structure of variable names
  - Started with a letter or an underscore
  - Followed by any numbers of letters, numbers, underscores
    >>>_1, a_, __, __, __a__, var0
    >>>0var, var@com, 1$var

• Python 3.0 reserved words
  None, False, True, from, import, nonlocal, global, not, and, or, in, lambda, def, return, class, del,
  if, else, for, while, break, pass, continue, try, except, with, as, finally, yield, assert

• In Python 2.6, print and exec are reserved words, nonlocal is not
Python naming conventions

- Variables whose names begin with a single underscore are not imported by a `from module import *` statement.
- Names that start and end with two underscores are system variables having special meanings, e.g., `__class__`.
- Names that start with two underscores and that do not end with two more underscores are localized to enclosing classes, e.g., `__x ==> _Class__x`.
• Python naming conventions
  - ..
    - Variable _ retains the result of the last expression in the interpreter
      >>> x = 1
      >>> x += 1
      >>> x
      2
      >>> _
      2
If statement 1

- **General Format**

```python
if <test1>:
    <statement1>
elif <test2>:
    <statement2>
elif <test3>:
    <statement3>
elif <test4>:
    <statement4>
else:
    <statement5>
```

```bash
>>> var = 1
>>> if var == 1:
    print(1)
    elif var == 2:
        print(2)
    elif var == 'str':
        print('str')
    else:
        print('nothing')
```
Python uses Indentation as Block delimiters. That is, all statements have the same indentation belong to the same block.

```python
>>> if 1 == 1:
    print(1)
    else:
    print('nothing')

>>> if 1 == 1: print(1) else: print('ok')
>>> if 1 == 1: print(1)
```
If statement 3

- Python uses a backslash to continue a line

```python
>>> if 1 == 1 and 2 == 2 and \
    3 == 3 and 4 == 4:
    print('that is fine')
else:
    print('syntax error')
```

- Python if/else Ternary Expression

```python
>>> A = 1 if 'ok' else 2
>>> A
1
```
• Two main looping structures in Python: while loop and for loop

```python
while <test>:
    <statements1>
else:
    <statements2>

for <target> in <object>:
    <statements1>
else:
    <statements2>
```

```python
>>> a,b=0,10
>>> while a<b:
    print(a, end=' ')  #what does this “end” mean?
    a += 1
0 1 2 3 4 5 6 7 8 9

>>> for x in range(10):
    print(x, end=' ')
0 1 2 3 4 5 6 7 8 9
```
Loop statement 2

- **break**
  - Jump out of the closest enclosing loop
- **continue**
  - Jump to the top of the closest enclosing loop
- **pass**
  - Do nothing. It is an empty statement
- **The loop else**
  - Run if and only if the loop is existed normally (without hitting a break)
```python
x=10
while x:
    x -=1
    if x % 2 != 0:
        continue
    print(x, end=' )
while True:
    name = input('your name?')
    if name=='stop':
        break
    age = input('your age?')
    print(name, ':', int(age))
x = y // 2
while x > 1:
    if y % x == 0:
        print(y, 'has factor', x)
        break
    X -= 1
else:
    print(y, 'is prime')
A
```
D = {'a':1, 'b':2}
for x in D:
    print(x, end=' ')

a b

D = {'a':1, 'b':2}
for (x,y) in D.items:
    print(x, ':', y, end=' ')

A a 1 b 2

T = [(1,2), (3,4), (5,6)]
for both in T:
    a,b = both
    print(a,b)

A 1 2
B 3 4
C 5 6
Parallel traversals: `zip` and `map`

- The built-in function `zip` allows to visit elements of multiple sequences in parallel.

```python
T1 = [1,2,3,4,5,6]
T2 = [10,20,30,40,50,60]
for x,y in zip(T1,T2):
    print(x,y)
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>30</td>
</tr>
<tr>
<td>D</td>
<td>40</td>
</tr>
<tr>
<td>E</td>
<td>50</td>
</tr>
<tr>
<td>F</td>
<td>60</td>
</tr>
</tbody>
</table>

```python
keys = ['a','b','c','d']
values = [10,20,30,40]
D = {}
for k,v in zip(keys,values):
    D[k] = v
AD = {'a':10, 'b':20, 'c':30, 'd':40}
```
Loop statement 6

- `map` in Python 2.6 can peer sequences in different lengths. The item-parallising function of `map` is not supported in Python 3.0

```python
T1 = [1,2]
T2 = [10,20,30]
map(None, T1, T2)

A = [(1, 10),(2, 20),(None,30)]
```
Python Functions

- What are functions?
  - Establishing a mapping between data (mapping from input data to output data)
  - Perform operations on data
- Why do we need functions?
  - Code encapsulation
  - Code reuse
  - Problem solving by “divide and conquer”
Python Functions

- **def** creates an object (a sequence of statements) and gives a name to this object – That is a Python function!
- **lambda** creates an un-named Python function
- **return** sends the output of a function to the caller.
- **yield** sends a value to the caller and suspend current status of the function
- **global** declares module-level variables
- **nonlocal** declares an appearance of an existing variable in the outer scope. Therefore, not local
- Arguments passed by reference
• `def` creates an object (a sequence of statements) and gives a name to this object – That is a Python function!

```python
def <name>(arg1,...,argN):
    <statement1>
    <statement2>
    ...
    <statementK>
```

```python
def <name>(arg1,...,argN):
    <statement1>
    <statement2>
    ...
    return <value>
```
Python `def` statement

- `def` is a statement, and can appear anywhere a statement can – even nested in other statements.

```python
if <test>:
    def func(arg1,...,argN):
        ...
else:
    def func():
        ...

Fname = func
Fname()

func.any_attr = "created by ..."
func.any_attr
A
B created by ...
```
Scopes 1

- The scope of a name refers to the location where this name can be looked up – the namespace of a variable
- How many different `var` in the follow code?

```python
if <test>:
    var = 1
    def func(arg1,...,argN):
        var = 2
        ...
else:
    var = 3
    def func():
        var1 = var
        ...
```
Scopes 2

• Rules of scopes
  - Names defined inside of a `def` creates a local scope, and are not visible outside of the codes of this `def`.
  - A module is a global scope, and spans a single file only
  - Each call to a function creates a new local scope
  - `global` declares a module-scope variable
  - `nonlocal` declares a variable visible in the enclosing function
  - All other names are enclosing function locals, globals, or built-ins
Scopes 3

Built-in

global(module)

Enclosing function locals

local(function)
Scopes 4

```
X = 100

def func1():
    X = 1
    print(X)

def func2():
    global X
    print(X)

#first.py
X = 100

#second.py
import first
print(first.X)

first.X = 0

#thismod.py
var = 99

def local():
    var = 0

def glob1():
    global var
    var += 1

Def glob2():
    var = 0
    import thismod
    thismod.var += 1

Def glob3():
    import sys
    glob = sys.modules['thismod']
    glob.var +=1
```
Scopes 5

- Nested scope
  - A reference `Var` firstly looks up `Var` in the current local scope, then from inner to outer in the local scopes of any enclosing functions, then in the global scope, and finally in the built-in scope.
  - `Var = value` creates the name in the current scope. If `Var` is declared by `global`, the scope of `Var` is the module-level; if `Var` is declared by `nonlocal`, the scope of `Var` is the closest enclosing function.
Scopes 6

```
X = 100
def func1():
    X = 88
def func2():
    print(X)
func2()

func1()
88

X = 100
def func1():
    X = 88
def func2():
    global X
    X = 90
    print(X)
func2()

func1()
90
```

```
def functional(N):
    def func(X):
        return X+N
    return func
F = functional(2)
F(3)
5
```
Scopes 7

- **lambda** expression introduces a new local scope

```python
def func():
    X = 88
    action = (lambda n: x+n)
    return action

X = func()
X(3)
```
Scopes with loop variables

- If an inner function is defined inside of a loop, and references a variable in the enclosing scope, all functions generated within the loop will have the value of the referenced variable in the last loop iteration.

```python
def functional():
    functions = []
    for I in range(5):
        functions.append(lambda n, I=I: I + n)
    return functions

F = functional()
F[0](2) = 2
F[2](2) = 4
F[4](2) = 6
```
Scopes 8

• Scopes with loop variables
  
  "If an inner function is defined inside of a loop, and references a variable in the enclosing scope, all functions generated within the loop will have the value of the referenced variable in the last loop iteration."

```python
def functional():
    functions = []
    for I in range(5):
        functions.append(lambda n: I + n)
    return functions

F = functional()
F[0](2) = 6
F[2](2) = 6
F[4](2) = 6
```
• Scopes with loop variables

```python
def functional():
    functions = []
    for I in range(5):
        functions.append(lambda n: I + n)
        print(functions[-1](2))
    return functions

F = functional()
2
3
4
5
6
F[0](2) = 6
F[2](2) = 6
F[4](2) = 6
```
Scopes 10

- nonlocal

```python
def test(start):
    state = start
    def nested(label):
        print(label, state)
        state += 1
    return nested

F = test(0)
F('hello')
```

```
UnboundLocalError: local variable 'state' referenced before assignment
```
Scopes 11

- nonlocal

def test(start):
    state = start
    def nested(label):
        nonlocal state
        print(label, state)
        state += 1
    return nested

F = test(0)
F('hello')
hello 0
F('world')
world 1
Scopes 12

- **Nonlocal**
  - Must have an outer enclosing function
  - Not look at module-level or built-in level

```python
def test(start):
    def nested(label):
        nonlocal state
        print(label, state)
        state += 1
        return nested

F = test(0)
F('hello')
```

UnboundLocalError: local variable 'state' referenced before assignment
Arguments of functions 1

- Argument assignment
  - Immutable arguments are passed by value
  - Mutable arguments are passed by reference

```python
def test(a,b):
    a = 10
    b[0] = 'changed'

X = 1
L = [1,2]
test(X,L)
X,L
(1, ['changed', 2])
```

```python
def test(a,b):
    a = 10
    b = b[:]
    b[0] = 'changed'

X = 1
L = [1,2]
test(X,L)
X,L
(1, [1, 2])
```
Arguments of functions 2

• Argument in function call
  - Match by position \( \text{func}(\text{value}) \)
  - Match by name \( \text{func}(\text{name}={\text{value}}) \)
  - Pass objects in a sequence \( \text{func}(*\text{seq}) \)
  - Pass key/value pairs in a dictionary \( \text{func}(**\text{dic}) \)

• Argument in function header
  - Matches any value by position \( \text{def func(\text{name})} \)
  - Default argument value \( \text{def func(\text{name}={\text{value}})} \)
  - Matches remaining args in a tuple \( \text{def func(*\text{name})} \)
  - Matches remaining key/values in a dic \( \text{def func(**\text{dic})} \)
Arguments of functions 3

```
def f(a, b, c):
    print(a, b, c)

f(1, 2, 3)
1 2 3

f(c=3, b=2, a=1)
1 2 3

f(1, c=3, b=2)
1 2 3

def f(a, b=2, c=3):
    print(a, b, c)

f(1)
1 2 3

f(c=6, a=1)
1 2 6

f(1, c=3, b=2)
1 2 3

def f(*a):
    print(a)

f(1, 2, 3)
(1, 2, 3)

f(1)
(1,)

def f(**a):
    print(a)

f(x=2)
{'x': 2}
```
def f(a, b, c):
    print(a, b, c)

arg = (1, 2, 3)
f(*arg)
1 2 3

A arg =
{'a':1, 'b':2, 'c':3}
f(**arg)
1 2 3

if <test>:
    func, arg=f1, (1,)
else:
    func, arg=f2,(1,2)
func(*arg)

---

def tracer(f,*arg1,**arg2):
    print('calling:',f.__name__)
    return f(*arg1,**arg2)

A def f(a,b,c,d):
    Return a + b + c + d
B
C print(tracer(f, 1,2,3,4))
D calling f
E 10
Python 3.0 keyword-only arguments

- They are named arguments after * arguments, not after ** arguments

```python
def func(a,*b,c):
    print(a,b,c)

func(1,2,c=3)
1 (2,) 3

func(a=1,c=3)
1 () 3

func(1,2,3)
TypeError: func() needs keyword-only argument c
```
Arguments of functions 5

```python
def func(a,**b,c):
    SyntaxError: invalid syntax
```

```python
def func(a,*b,**c,d=3):
    SyntaxError: invalid syntax
```

```python
def func(**a,*b, c):
    SyntaxError: invalid syntax
```

Test and find the rules yourself
"Print function
Call signature: print30(*args, sep=' ', end='\n', file=None)"

import sys

def print30(*args, **kargs):
    sep = kargs.get('sep', ' ')
    end = kargs.get('end', '\n')
    file = kargs.get('file', sys.stdout)
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
“Print function
Call signature: print30(*args, sep=' ', end='
', file=None)
”

import sys

def print30(*args, sep=' ', end='
', file=sys.stdout):
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)

print30(3, name='bob')
TypeError: print30() got an unexpected keyword argument 'name'
import sys

def print30(*args, **kargs):
    sep = kargs.pop('sep', ' ')
    end = kargs.pop('end', '
')
    file = kargs.pop('file', sys.stdout)
    if kargs:
        raise TypeError('extra keyword: %s' % kargs)
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)